GRAPHiC: Utilizing Graph Structures and Class Weights in Code Comment Classification with Pretrained BERT Models

Pir Sami Ullah Shah
National University of Computer and Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
samiullah.shah@nu.edu.pk

Shahela Saif
National University of Computer and Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
shahela.saif@nu.edu.pk

Muhammad Haris Athar
National University of Computer and Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
i211159@nu.edu.pk

Muhammad Riyaan Tariq
National University of Computer and Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
i211101@nu.edu.pk

Abdur Rehman Afzal
National University of Computer and Emerging Sciences (FAST-NUCES)
Islamabad, Pakistan
i211151@nu.edu.pk

***Abstract*—Source code comments are an essential part of the software development process, and the classification of these comments into relevant categories is crucial for code maintenance. For this problem, we present *GRAPHiC*, a set of classifiers designed for multi-label classification of source code comments in Java, Python, and Pharo. As part of *GRAPHiC*, we train three separate classifiers on the *NLBSE Code Comment Classification* dataset, using GraphCodeBERT and incorporate class weights to address dataset imbalance. The classifiers achieve an average F1 score of 0.71, outperforming the *SetFit* baseline score of 0.63 by 12%. This paper highlights the effectiveness of GraphCodeBERT for code comment classification and explores areas for further research. The models and training code are publicly available to facilitate replication and further experimentation.**

***Index Terms*—GraphCodeBERT, Code Comments, Multi-label classification, Transformer-Based Models, Class Weighting, NLP**

I. Introduction

While source code comments are an essential part of the software development process, their classification into relevant categories is equally important to ensure they serve their intended purpose [5]. Multi-label classification of these comments can help developers gain a better understanding of the comments and improve code maintenance. Due to the complex nature of code and comments, traditional models often struggle to accurately classify them, as demonstrated by Indika et al. [6]. In recent years, transformer-based models, which benefit from training on large code corpora, have shown improved performance in the classification of code comments [7].

In this research, we introduce ***GRAPHiC***, a graph-based transformer model for multi-label classification of code comments. GRAPHiC leverages `GraphCodeBERT`, a pretrained model for programming language tasks [8], to classify comments in Java, Python, and Pharo. To address dataset imbalance, such as the significant disparity in Java's *Deprecation* label (117 positive vs. 7497 negative instances), we employ a class weighting strategy [9]. GRAPHiC outperforms the SetFit baseline provided by the competition organizers, achieving an average F1 score of 0.71 across all languages. Our trained models [10] and code [11] are publicly available.

II. Related Work

Various studies have explored techniques for code comment classification. Pascarella et al. [4] developed a taxonomy for classifying Java code comments, highlighting the role of ML in generalizing across datasets, which laid the groundwork for applying similar methods to other languages.

Building on this foundation, Rani et al. [2] proposed an automated approach for classifying comment categories in Java, Python, and Smalltalk using text analysis and natural language processing (NLP).

Al-Kaswan et al. [3] introduced STACC, a framework developed for the NLBSE 2023 competition. It utilized `SentenceTransformers`-based binary classifiers for Java, Pharo, and Python. Their framework significantly improved the F1 score from a baseline of 0.31 to 0.74, demonstrating the effectiveness of `SentenceTransformers`.

In 2024, Hai et al. [7] presented Dopamin, a `Transformer`-based tool that improved comment classification performance by 3% over STACC through

domain-specific post-training and multi-level layer aggregation.

Unlike previous work, which focused on binary classification, our research targets multi-label classification for Java, Python, and Pharo. Like Dopamin [7], we also employ a `Transformer`-based model to address the inherent complexity of our topic.

III. Tool Construction

This section outlines the approach for implementing GRAPHiC. It is divided into three parts detailing the steps taken for the development of robust classifiers.

A. Dataset Stratification

The dataset exhibits an imbalanced label distribution, with certain labels being significantly more frequent than others.

TABLE I
Label Distribution across all three Languages

Java	**Pos (Train)**	**Neg (Train)**	**Pos (Test)**	**Neg (Test)**
Summary	3610	4004	892	833
Ownership	267	7347	45	1680
Expand	509	7105	102	1623
Usage	2093	5521	431	1294
Pointer	904	6710	184	1541
Deprecation	117	7497	15	1710
Rational	311	7303	68	1657
Python	**Pos (Train)**	**Neg (Train)**	**Pos (Test)**	**Neg (Test)**
Usage	578	1306	121	285
Parameters	572	1312	128	278
DevNotes	210	1674	41	365
Expand	343	1541	64	342
Summary	347	1537	82	324
Pharo	**Pos (Train)**	**Neg (Train)**	**Pos (Test)**	**Neg (Test)**
KeyImplPts	178	1120	43	246
Example	547	751	119	170
Resp	245	1053	52	237
ClassRefs	46	1252	4	285
Intent	151	1147	30	259
KeyMsgs	214	1084	43	246
Collaborators	76	1222	10	279

To facilitate robust model selection, we stratify 20% of the dataset for each language using a stratification key. This key is calculated as the sum of the label values for each instance. For example, a Java instance with labels [1, 0, 1, 0, 0, 1, 0] has a key of 3, representing the total active labels. This approach ensures proportional representation of instances with similar label counts, preserving the label distribution for balanced model selection. The stratified subset is then split into 80% training and 20% testing sets.

B. Model Selection

For the implementation of GRAPHiC, we consider three transformer-based models: `GraphCodeBERT` [8], `CodeBERT` [12], and `CodeBERTa` [13]. `CodeBERT`, tailored for source code, is trained on a large code corpus and is well-suited for code-related tasks like classification and summarization. `CodeBERTa` is a variant of `CodeBERT`, designed to be lighter and potentially faster, though its reduced number of parameters might limit its performance in certain tasks [13]. `GraphCodeBERT` incorporates a graph-based structure to understand different parts of the code beyond plain text, which aids in tasks requiring structured code understanding [8]. These models are selected to ensure strong performance in multi-label classification of code comments [14].

C. Class Weights

The class imbalance, as referenced in Table I, poses a significant challenge for training the model effectively. To address this, we employ a class weighting strategy [9], where class weights for each category are calculated as the inverse of their class frequencies. This approach assigns higher weights to the less frequent classes and lower weights to the more frequent ones, ensuring that the poorly performing classes receive more attention during training:

$$W_i = \frac{1}{\text{Frequency of Class } i} \tag{1}$$

This strategy mitigates the impact of class imbalance and improves model performance on underrepresented classes.

IV. Experimental Setup

A. Dataset

We use a labeled dataset provided for the NLBSE 2025 Tool Competition,[1] comprising 14,875 code comment sentences across three languages (Java, Python, and Pharo) from Hugging Face. Each row represents a sentence assigned to multiple labels, depending on the specific attributes and descriptions of the code snippet it accompanies. The dataset is split into six parts, following a training-testing split for each language.

B. Metrics

The metrics we use to evaluate our tool's performance are P_c (Precision), R_c (Recall), and $F1_c$ (F1 score). These scores are calculated per category c, where F1 is the harmonic mean of Precision and Recall. The metrics are defined as follows:

$$P_c = \frac{TP_c}{TP_c + FP_c}, \quad R_c = \frac{TP_c}{TP_c + FN_c}, \quad F1_c = 2 \cdot \frac{P_c \times R_c}{P_c + R_c} \tag{2}$$

Additionally, we calculate the overall submission score as:

$$\text{score} = 0.6 \cdot \text{avg F1} + 0.2 \cdot \frac{5 - \text{runtime}}{5} + 0.2 \cdot \frac{5000 - \text{GFLOPS}}{5000} \tag{3}$$

The submission score ensures holistic evaluation by balancing accuracy, efficiency, and scalability with non-negative constraints for runtime and FLOPs.

C. Baselines

The SetFit baseline for this project, provided by the competition organizers, is based on `Sentence Transformers`. Analysis of the baseline results, as seen in Table IV, reveals significant variability in F1 scores across categories, which we attribute to the imbalanced label distributions in the dataset. For example, categories like Java's *Deprecation* and Pharo's *Class References*, with very few positive instances, exhibit poor F1 scores, highlighting the need for strategies to improve performance in challenging categories with unbalanced data.

[1]Dataset: https://huggingface.co/datasets/NLBSE/nlbse25-code-comment-classification

D. Implementation

We implement this experiment using the `Hugging Face Transformers`[2] library and `PyTorch`[3]. We preprocess the dataset using `Pandas` and the Hugging Face Dataset library and load the dataset directly from Hugging Face. The experiment is executed on Google Colab, which provides access to a T4 GPU that we employ for training and testing.

V. Results and Analysis

A. Dataset Stratification

After applying stratification, the dataset is divided into training and test sets for each programming language. These

TABLE II
SIZE OF THE STRATIFIED DATASET

Language	Training Set	Test Set
Java	1,218	305
Python	301	76
Pharo	208	52

stratified splits ensure balanced label distributions across the subsets for consistent evaluation of the candidate base models.

B. Model Selection

The results of the model selection experiments, trained on the stratified dataset, are presented in Table III. From the table, it is clear that `GraphCodeBERT` outperforms both `CodeBERTa` and `CodeBERT` by a significant margin in terms of the Average F1 Score. This is likely due to its ability to leverage graph structures [8], allowing it to capture complex relationships between code comments.

TABLE III
MODEL SELECTION RESULTS

Model	Average F1 Score
GraphCodeBERT	0.5830
CodeBERTa	0.5109
CodeBERT	0.4797

C. Class Weighting

The class weights, calculated as the inverse of class frequencies, give more influence to minority classes during training [9]. For example, Pharo's *Collaborators* and *ClassReferences* receive higher weights due to their lower frequencies. This weighting approach ensures that minority classes are adequately prioritized, reducing bias toward majority classes.

D. Hyperparameters

We choose the hyperparameters–batch size (4) and epochs (6 for Java, 11 for Python, 12 for Pharo)–based on practical constraints and empirical testing. We set the random seed to 42, following a common convention in machine learning, to ensure reproducibility using `set_seed`, `torch.manual_seed`, `np.random.seed`, and `random.seed`. Further hyperparameter optimization is not explored due to time limitations.

[2]https://huggingface.co/docs/transformers/index
[3]https://pytorch.org/

E. GRAPHiC Models

We train a classifier for each of the three languages, with the training process taking approximately 1 hour. The results of training GRAPHiC, as presented in Table IV, reveal that the average F1 score of our tool, 0.71, is higher than that of the SetFit baseline's 0.63. With the inclusion of class weights, GRAPHiC consistently outperforms SetFit in the $Precision$ and $F1$ metrics, and notable performance gains are observed in categories like Python's *DevelopmentNotes* and *Expand*, which are particularly challenging for the baseline models. However, the model performs worse in $Recall$ for certain categories, highlighting areas for potential optimization.

The inference runtime for GRAPHiC is higher than SetFit's due to its greater computational complexity from fine-tuning a `transformer`-based model. The *0* in Equation 3 reflects the non-negative constraint on runtime, emphasizing the balance between model complexity and efficiency. Per Equation 3, we achieve a final submission score of:

$$0.71 * 0.6 + (0) + (0.2 * (5000 - 3245)/5000) = \mathbf{0.5}$$

F. Analysis and Discussion

When comparing languages, our tool performs best on Java, followed by Python and Pharo. The average F1 scores across the three languages are 0.74, 0.69, and 0.68, respectively. These scores are influenced by the dataset sizes and imbalances, as Java has the largest dataset, followed by Python and Pharo. Our tool significantly outperforms the SetFit baseline in F1 scores, demonstrating `GraphCodeBERT`'s ability to model relationships within textual data effectively.

A significant challenge in model training was the dataset imbalance. The addition of class weighting played a critical role in tackling this problem. This is evident in the increased F1 scores for Java's *Rational* (from 0.20 to 0.34), Python's *Expand* (from 0.55 to 0.66), and Pharo's *Collaborators* (from 0.36 to 0.47), as seen in Table IV. These improvements highlight `GraphCodeBERT`'s strengths, as its pretraining allows it to understand relationships between phrases by capturing semantics and structural context within comments [8].

However, Java's *Ownership* and *Deprecation* classes-despite having fewer positive instances than *Rational*-performed well in both the baseline and our implementation. The better performance of *Ownership* can be attributed to its simpler, more identifiable patterns, like the consistent use of the $@author$ tag, which aids classification. Similarly, *Deprecation* comments often include version information and specific deprecation annotations (e.g., $@deprecated$ and $@since$), providing clear signals for the model to recognize. In contrast, *Rational* may involve more abstract annotations, making it harder for the model to generalize.

The lower Recall in the model can be attributed to the class imbalance, particularly in categories with a large disparity between positive and negative instances. For example, Pharos's *Collaborators* category has 76 positive and 1,222 negative instances, limiting the model's ability to correctly identify

TABLE IV
SCORES COMPARISON: SETFIT BASELINE, GRAPHIC, AND DIFFERENCES

Lang	Category	SetFit Baseline			GRAPHiC			Differences
		Precision	Recall	F1	Precision	Recall	F1	F1
Java	Summary	0.87	0.82	0.85	0.90	0.90	0.90	+0.05
	Ownership	1.00	1.00	1.00	1.00	1.00	1.00	0.00
	Expand	0.32	0.44	0.37	0.45	0.43	0.44	+0.07
	Usage	0.91	0.81	0.86	0.93	0.84	0.88	+0.02
	Pointer	0.73	0.94	0.82	0.77	0.97	0.86	+0.04
	Deprecation	0.81	0.60	0.69	0.84	0.73	0.78	+0.09
	Rational	0.16	0.29	0.20	0.31	0.38	0.34	+0.14
Python	Usage	0.70	0.73	0.71	0.84	0.74	0.78	+0.07
	Parameters	0.79	0.81	0.80	0.84	0.82	0.83	+0.03
	DevNotes	0.24	0.48	0.32	0.34	0.36	0.35	+0.03
	Expand	0.43	0.76	0.55	0.62	0.70	0.66	+0.11
	Summary	0.64	0.58	0.61	0.77	0.86	0.81	+0.20
Pharo	KeyImplPts	0.63	0.65	0.64	0.70	0.60	0.65	+0.01
	Example	0.87	0.90	0.88	0.94	0.91	0.93	+0.05
	Responsibilities	0.59	0.59	0.59	0.61	0.63	0.62	+0.03
	ClassRefs	0.20	0.50	0.28	0.50	0.75	0.60	+0.32
	Intent	0.71	0.76	0.74	0.78	0.86	0.82	+0.08
	KeyMsgs	0.68	0.79	0.73	0.70	0.79	0.74	+0.01
	Collaborators	0.26	0.60	0.36	0.57	0.40	0.47	+0.11
Average		0.6112	0.6906	0.6359	**0.7097**	**0.7226**	**0.7120**	

positive instances. This imbalance results in reduced Recall across several categories.

Overall, `GraphCodeBERT`'s graph embeddings provide significant advantages in capturing semantic and structural patterns within code comments. However, challenges remain with underrepresented classes, requiring further strategy optimization to fully harness its capabilities in text-based multi-label classification tasks.

VI. CONCLUSION

In this study, we present GRAPHiC, a tool for multi-label classification of code comments in Java, Python, and Pharo. GRAPHiC utilizes the graph-based architecture of its base model to capture structural relationships within code comments, enabling it to outperform the SetFit baseline in terms of F1 scores. Notably, it excels in classifying comments related to *Summary* (Java) and *Usage* (Python). However, the performance of GRAPHiC shows room for improvement in certain categories. Its overall performance varies across languages, with the lowest observed in Pharo, underscoring the need for further adaptation to datasets with limited data. Our model, with a 12% increase over the SetFit baseline in terms of F1, is publicly available for reproducibility [10].

REFERENCES

[1] Ali Al-Kaswan, Giuseppe Colavito, Nataliia Stulova, and Pooja Rani. The NLBSE'25 Tool Competition. In *Proceedings of The 3rd International Workshop on Natural Language-based Software Engineering (NLBSE'25)*, 2025.

[2] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz, "How to identify class comment types? A multi-language approach for class comment classification," *Journal of Systems and Software*, vol. 181, p. 111047, 2021, doi: https://doi.org/10.1016/j.jss.2021.111047.

[3] A. Al-Kaswan, M. Izadi, and A. Van Deursen, "STACC: Code comment classification using SentenceTransformers," in *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, 2023, pp. 28–31, doi: 10.1109/NLBSE59153.2023.00014.

[4] L. Pascarella and A. Bacchelli, "Classifying code comments in Java open-source software systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, doi: 10.1109/MSR.2017.34.

[5] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, "A decade of code comment quality assessment: A systematic literature review," *Journal of Systems and Software*, vol. 195, p. 111515, 2023, doi: https://doi.org/10.1016/j.jss.2022.111515.

[6] A. Indika, P. Y. Washington, and A. Peruma, "Performance Comparison of Binary Machine Learning Classifiers in Identifying Code Comment Types: An Exploratory Study," in *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, 2023, pp. 20–23, doi: 10.1109/NLBSE59153.2023.00012.

[7] N. L. Hai and N. D. Q. Bui, "Dopamin: Transformer-based comment classifiers through domain post-training and multi-level layer aggregation," in *2024 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*, 2024, pp. 61–64.

[8] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," arXiv, 2021. [Online]. Available: https://arxiv.org/abs/2009.08366.

[9] B. Bakırarar and A. ELHAN, "Class Weighting Technique to Deal with Imbalanced Class Problem in Machine Learning: Methodological Research," in *Turkiye Klinikleri Journal of Biostatistics*, vol. 15, 2023, pp. 19–29, doi: 10.5336/biostatic.2022-93961.

[10] H. Athar, "Hugging Face Collection for GRAPHiC," 2025. [Online]. Available: https://huggingface.co/collections/harisathar04/graphic-nlbse-675b2addd2424f2994df1547

[11] H. Athar, "GRAPHiC Repository," 2025. [Online]. Available: https://github.com/harisathar04/graphic-nlbse25.git

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *arXiv*, 2020, doi: 10.48550/arXiv.2002.08155.

[13] Hugging Face, "CodeBERTa-small-v1," *Hugging Face Hub*, 2024. [Online]. Available: https://huggingface.co/huggingface/CodeBERTa-small-v1

[14] K. Liu, G. Yang, X. Chen, and Y. Zhou, "EL-CodeBert: Better Exploiting CodeBert to Support Source Code-Related Classification Tasks," in *Proceedings of the 2022 ACM SIGPLAN International Workshop on the State of the Art in Program Synthesis*, 2022, pp. 147–155, doi: 10.1145/3545258.3545260.